Spring 5-1-2012

Extending the Functionality of a Compiler for Linear Algebra Optimization

Pavel Zelinsky
University of Colorado Boulder

Pavel Zelinsky

Extending the Functionality of a Compiler for Linear Algebra Optimization

Senior Thesis

May 1st, 2012

1. Introduction

Large scale scientific applications take a significant amount of time to run. Optimizing these applications is vital for reducing the time and the cost of running these applications. At the core of computations, these applications often use linear algebra operations on large matrices and vectors. Optimizing linear algebra kernels by hand is a tedious and lengthy process, so often times highly tuned libraries are used to simplify the process. Rapid hardware development consistently makes such libraries outdated though. Auto-tuning, which is when a compiler or some similar tool optimizes code

for the user, is able to incorporate a range of optimizations for any particular given hardware. We have created a compiler that automates optimizing such linear algebra kernels for large scale scientific applications. Our compiler uses a mix of partitioning, cache tiling, and loop fusion to auto-tune code. In this paper I discuss the issues that the compiler aims to fix, my updates to the compiler, and why these updates are useful.

2. Memory Hierarchy

For years, processor speeds have risen more than memory access speeds[7], and this has lead to the processor having to wait on memory accesses to perform any operations. It is important to optimize scientific applications in terms of memory access patterns. This section discusses the structure of the memory that can be taken advantage of in order to minimize main memory accesses.

The Cache

Processor vendors have started adding caches that store small amounts of data closer to the processor and have significantly higher access speed than the main memory. Compared to a single floating point operation, accessing the L1 cache is an order of magnitude slower, with the L2 cache being several orders of magnitude slower, and a single memory access being around a million times slower[3]. In other words, for operations on large matrices, even the L2 cache is not big enough, and data must be stored in main memory. Performance is thus limited by data movement from main memory.

The way that the cache works is when something is accessed in main memory, instead of pulling in a singular piece of data, the cache pulls in a chunk of data and stores it closer to the processor[7]. This chunk of data fills a single line of cache, and the cache has many cache lines,

depending on the vendor. This way, future data accesses near the original data accessed are done through the cache instead of through main memory, making future accesses potentially faster. Therefore, it is important to consider data reuse and sequential data access when performing linear algebra, otherwise the cache is not efficiently used.

Thrashing

Thrashing is an important consideration when optimizing code for cache reuse. Thrashing occurs when data is no longer in the cache because it is constantly being recycled, due to data accesses[7]. Let's say, for example, a computer has two lines of cache and is accessing three vectors, alternating which vector is accessed each time. As the first and second vectors are accessed, each line of cache gets a vector. But when the third vector is accessed, it replaces the first vector in the first line of cache. When the first vector is accessed again, it replaces the second vector, when the second vector is accessed it replaces the third vector, and so on. All vectors get accessed sequentially yet the cache is unable to fit the vectors as they are accessed, and thrashing occurs. As a result, preventing cache thrashing plays a significant role in cache reuse for code optimization.

Spilling

During compilation, commonly used variables are assigned to registers so that the registers are used optimally in run time. When there are not enough registers for all the commonly used variables, some of the variables are stored in memory and must be swapped into the registers when used. This is register spilling, and it reduces performance since instead of only using the registers, in portions of code it has to access memory. The way to counteract register spilling is either to increase the number of registers with new hardware, or to avoid using more variables at any particular time than there are registers available.

3. Linear Algebra

At the core of many scientific programs are linear algebra kernels, so optimizing these linear algebra kernels is vital for high performance. This section discusses several approaches to optimizing linear algebra operations as well as possible weaknesses of these approaches.

Linear Algebra Libraries

In order to help produce highly optimized linear algebra kernels, several linear algebra libraries have been made. Having a library to call makes writing efficient code easier without having to study optimizations. The BLAS, or the Basic Linear Algebra Subprograms, is one of these libraries, a basic Fortran version of which can be obtained at www.netlib.org. This library has all basic matrix and vector operations, including matrix-vector multiplication, vector dot product, and matrix addition. However, this is not optimized code. There are versions of the BLAS tuned for performance by hardware vendors. A particular example of vendor tuned BLAS is in the Math Kernel Library (MKL)[8]. MKL is optimized for Intel hardware only. It contains the BLAS, as well as commonly used routines like eigenvalue problems or QR factorization. Similarly there is Goto BLAS[9], a version of the BLAS tuned for performance on any hardware. Goto BLAS has no extra routines outside of the BLAS like MKL. LAPACK (Linear Algebra PACKage) has many of the same commonly used routines that are in MKL but not in BLAS. All of these libraries are primarily meant for dense matrices only.

```
for(i=0;i<n;i++)
    for(j=0;j<n;j++)
        x[i]+=A[i][j]*b[j]
for(i=0;i<n;i++)
    for(j=0;j<n;j++)
        y[i]+=A[i][j]*c[j]
```

Figure 1: Two matrix-vector products, unoptimized

Linear algebra libraries help the memory inefficiency problem since a particular routine can be optimized for memory use. The unfortunate part is when there are successive library calls. Although each routine on its own is highly optimized, the routines as a whole are not as optimized as they could be. This

can be shown through two matrix-vector products that are using the same matrix. A library performs the two operations separately, thus there are two separate doubly nested loops to access the matrix (Figure 1). In other words, the matrix is read in from memory twice to perform the operations. Yet if the two loops are combined into a single loop, the matrix is read from memory once (Figure 2), providing significant speedups[10]. As a result, current linear algebra libraries are not ideal when looking at a full scientific application.

```
for(i=0;i<n;i++)
    for(j=0;j<n;j++)
        x[i]+=A[i][j]*b[j]
        y[i]+=A[i][j]*c[j]
```

Figure 2: Two matrix-vector products, optimized

To fix this issue, we can build a new library with commonly used BLAS combinations. Instead of calling a single matrix-vector product twice, a user calls a single routine that performs the two matrix-vector products, fixing the memory issue for the particular routine combinations that are encompassed in such a library. In previous research of the LAPACK routines, I showed that there are no emerging patterns in successive BLAS calls, despite these calls being a common occurrence. Due to this, writing a library of BLAS call combinations is unfeasible since there are too many combinations to consider.

```
'   Input: x (vector)
    Output: v (vector) , β (scalar)
    n = length(x)
    sig = x(2:n)^T x(2:n)
    v = [ 1 x(2:n) ]
    if sig = 0
            β = 0
    else
            μ = sqrt(x(1)^2 + sig)
            if x(1) <= 0
                    v(1) = x(1) − μ
            else
                    v(1) = -sig/(x(1) + μ)
            end
            β = 2v(1)^2/(sig + v(1)^2)
            v = v/v(1)
    end
```

Figure 3: Algorithm for finding the Householder Vector

$$\text{In}: A, u, z, u_0, z_0, w_0, v_0, \alpha, \beta$$
$$\text{Out}: B, x, w$$
$$w \leftarrow 0$$
$$\text{For i} = 1:b,$$
$$B_i \leftarrow A_i + u_0 z_{0i}^T$$
$$B_i \leftarrow B_i + w_0 v_{0i}^T$$
$$x_i^T \leftarrow \beta u^T B_i + z_i^T$$
$$w \leftarrow w + \alpha B_i x_i$$
$$\text{End for}$$

Figure 4: Pseudo code for GEMVER

$\text{In} : A, u, z, \alpha, \beta$
$\text{Out} : x, w$
$w \leftarrow 0$
For $\text{i} = 1 : b,$
$\quad x_{\text{i}}^{\text{T}} \leftarrow \beta u^{\text{T}} A_{\text{i}} + z_{\text{i}}^{\text{T}}$
$\quad w \leftarrow w + \alpha A_{\text{i}} x_{\text{i}}$
End for

Figure 5: Pseudo code for GEMVT

$Input : A(matrix)$
$Output : B(matrix)$
$for \quad j = 1 : n$
$\quad [v, \beta] = house(A(j : m), j))$
$\quad A(j : m, j : n) = (I_{m-j+1} - \beta v v^T) A(j : m, j : n)$
$\quad A(j+1 : m, j) = v(2 : m - j + 1)$
$\quad if \quad j \le n - 2$
$\quad\quad [v, \beta] = house(A(j, j+1 : n)^T)$
$\quad\quad A(j : m, j+1 : n) = A(j : m, j+1 : n)(I_{n-j} - \beta v v^T)$
$\quad\quad A(j, j+2 : n) = v(2 : n - j)^T$
$\quad end$
end

Figure 6: Householder bidiagonalization

Householder bidiagonalization

Householder bidiagonalization[1] is a particular example of a possible kernel at the core of a scientific application. It is an algorithm to reduce any matrix to a bidiagonal form through a series of matrix products. This is useful to do since if the matrix is used in future operations, performing operations on a bidiagonal matrix is significantly cheaper than on a dense matrix. In order to zero out elements and get the matrix into a bidiagonal form the algorithm uses Householder vectors. Finding a Householder vector takes O(n) operations, and the algorithm is in Figure 3. The Householder vector algorithm (Figure 3) takes in a row or column of the matrix to be bidiagonalized as an input, and outputs the Householder vector and a scalar. Taking the original matrix and multiplying it by ($I - \beta v^T v$), where the vector v is a Householder vector for a particular row or column, makes all the terms in that row or column zero but the two diagonals. It is important to note that calculating a Householder vector requires normalizing a vector, and thus requires square root and division. The rest of the Householder Bidiagonalization algorithm takes $O(n^3)$ operations[1], and the algorithm is shown in Figure 6, where the house function returns the Householder vector for the input vector. In Figure 6, we can see the Householder vectors being applied first on the left side, then on the right side, of submatrices of the

original matrix. Any time that we apply a Householder vector on the left side, a column of the submatrix becomes all zeroes except the first element, and any time that we apply a Householder vector on the right wide, a row of the submatrix becomes all zeroes except the first element. Although the entire row and column becomes zeroes, the overall result is a bidiagonal matrix because the submatrices are not always accessed as square matrices. Although we are using the submatrices, this is still a factorization on the original matrix, because any Householder vector for the full matrix is simply all zeroes except the bottom terms are the Householder vector from a particular submatrix.

By introducing two new operations, GEMVER (Figure 4) and GEMVT (Figure 5), Householder bidiagonalization can be optimized for efficient memory use[5]. In this optimized algorithm, GEMVER is used to perform all the matrix updates. This can be seen since GEMVER calculates the outer products and uses them to scale the matrix B. The way the optimization works is by deferring updates to the matrix. We can see this in GEMVER, since in the basic algorithm for Householder Bidiagonalization we the original matrix is multiplied by $(I - \beta v^T v)$, but in GEMVER the matrix is multiplied by a combination of Householder vectors simultaneously to reduce the number of reads and writes in the matrix. By putting the core of the computation within GEMVER and GEMVT, the algorithm reduces cache conflicts and matrix accesses, and produces significant speed ups.

Square Root and Division Implementations [4]

When using an implementation of square root and division the main thing to consider in terms of this project is accuracy. When doing error checking we must know what the error bounds are at the end of all the computations. There are two main multiplicative algorithms for division and square root, the Newton-Raphson method (Figure 7)

$$x_{i+1} = x_i \times (2 - b \times x_i)$$
$$x_{i+1} = 1/2 \times x_i \times (3 - a \times x_i^2).$$

Figure 7: Newton-Raphson method for finding 1/b and square root of a

Division

$$x_{i+1} = x_i \times r_i \text{ and } y_{i+1} = y_i \times r_i.$$
$$r_i = 2 - y_i$$

Square root

$$x_{i+1} = x_i \times r_i^2 \text{ and } y_{i+1} = y_i \times r_i,$$
$$r_i = (3 - x_i)/2$$

Figure 8: Goldschmidt's algorithms for division and square root

and Goldschmidt's algorithm (Figure 8). The Newton-Raphson method for finding 1/b is obtained by applying Newton's Method for root finding on the function f(x)=b-1/x. By plugging in x=1/b, we can see that f(1/b)=0 and that in Figure 7 the algorithm for division has x_n converge to 1/b. The Newton-Raphson method for finding the square root of a is obtained by applying Newton's Method on the function $f(x)=a-1/x^2$. These are all iterative algorithms, with x_0 being a seed that is a guess at the value of the division or the square root. In Goldschmidt's algorithm for division, x converges to a/b and y converges to 1, while in his algorithm for square root y converges to the square root of a and x converges to 1. Both of these algorithms converge at a quadratic rate.

Sparse Matrices

Sparse matrices are matrices with mostly zero elements. Since most of the matrix is zeroes, it is inefficient to store the entire matrix. Because of this it is important to understand different sparse matrix storage formats. The two formats that work well for any sparse matrix are CSR (Compressed Sparse Row) and CSC (Compresses Sparse Column)[14]. The difference between the two storage formats is whether the matrix is being interpreted as a row of columns or a column of rows. CSR defines the matrix using three vectors. The first vector is of each non-zero value in the matrix, going through the matrix row by row. The second vector is indexes of the non-zero elements within each row, and the third vector is to handle the column indexes of each non-zero element. The way that the third vector works is by saying where each row starts, so when the second vector goes to the next row it uses the third vector to know where the next row of non-zero elements is. CSC is the same as CSR, except the first matrix is of the non-zero elements going through the matrix column by column, the second vector is the indexes within each column, and the third vector is to handle the row indexes of each non-zero element.

There are many other matrix-specific formats. For example a bidiagonal matrix is stored as a single vector of only the two diagonals since the location of each non-zero element is known

beforehand. There are formats that are a mix of storing the diagonal in a single vector and CSR or CSC for the rest of the matrix. Another interesting format to look at is when instead of storing a single value of the matrix in CSR or CSC, you store a small matrix at each location. This format only makes sense if the matrix is known to have dense blocks of values within the sparse matrix.

4. Memory Optimizations

Memory optimizations prioritize minimizing data movement to and from main memory by maximizing cache reuse. There are two main memory optimizations of interest: loop fusion and vector interleaving. Both of these optimizations reduce the number of times matrices must be read from or written to main memory.

Loop Fusion

Loop fusion works by combining inner and/or outer loops of operations. If, for example, we want to perform two matrix-vector products using two different vectors but the same matrix, we can combine the inner loops to use each element in the matrix twice once the element is read from main memory (Figure 9). On the other hand, if we have a matrix-vector multiply followed by a vector dot product, it is only possible to combine the two outer loops. From looking at the effects of loop fusion, it would

No optimizations:

```
for(i=0;i<n;i++)
    for(j=0;j<n;j++)
        x[i]+=A[i][j]*b[j]
for(i=0;i<n;i++)
    for(j=0;j<n;j++)
        y[i]+=A[i][j]*c[j]
```

Loop Fusion:

```
for(i=0;i<n;i++)
    for(j=0;j<n;j++)
        x[i]+=A[i][j]*b[j]
        y[i]+=A[i][j]*c[j]
```

Vector Interleaving:

```
for(i=0;i<n;i++)
    temp_in[2*i]=b[i]
    temp_in[2*i+1]=c[i]
for(i=0;i<n;i++)
    for(j=0;j<n;j++)
        temp_out[2*i]+=A[i][j]
                    *temp_in[2*i]
        temp_out[2*i+1]+=A[i][j]
                    *temp_in[2*i+1]
```

Overhead

Figure 9: Memory optimizations

appear that fusing as much as possible is optimal, however this is not the case. For example, if there are enough matrix-vector products in a row and everything is fully fused, there are not enough registers to hold all the needed values[15]. Not having enough registers for all the values forces the computer to put values on the stack instead of the registers, so not to the fastest memory and slows down the program. Because of the possibility of register spilling or even worse thrashing the cache, in order to find the most efficient way to do loop fusion we must look over all possible loop fusions. For small problems like several matrix-vector products this is simple, but for larger kernels becomes infeasible to do by hand.

Vector Interleaving

Vector interleaving takes the two vectors used in multiplications and combines them into one large vector with alternating values from the two originals[11][12]. Interleaving the vectors adds extra operations (marked as overhead in Figure 9), but the number of operations added grows linearly with the order of the matrix since the vector length grows linearly with the order of the matrix. The overhead is small as compared to the quadratic growth of the rest of the code. After combining the two vectors into the interleaved vector, the multiplications are combined into a single loop. The combination of operations into a single loop makes vector interleaving similar to loop fusion in the way the matrix is accessed, which can be seen in Figure 9 since both optimizations access A[i][j] twice for each i and j in the same order. Vector interleaving additionally increases the locality of the vector accesses for the operations since any time that one vector is accessed, the other is as well, which in an interleaved vector is two successive accesses. Finding the ideal way to interleave vectors is a problem similar to that of finding the ideal set of loop fusions, and with large enough kernels is infeasible to do by hand.

Dense Matrices

Since vector interleaving and loop fusion both optimize code in similar ways, it is important to consider which produces more speedups. I randomly generated dense matrices of different sizes, and did performance testing on loop fusion and vector interleaving on a series of matrix vector multiplications. Figure 10 shows the results of this experiment when run on an AMD Opteron machine, with a 2.6 GHz processor, 64 KB L1 cache, and a 1 MB L2 cache with the -O3 flag. As we can see, vector interleaving (green line) performs better than full loop fusion (red line) only in very particular scenarios. For most cases, vector interleaving is worse or about the same as loop fusion. For the particular machine I ran the tests on, vector interleaving outperforms loop fusion at five matrix vector multiplications, which happens because with enough vectors, loop fusion begins to have memory issues from register spilling. Vector interleaving being slower than loop fusion is expected, since the benefit they provide is similar in both cases but vector interleaving introduces an overhead. Any performance gains from vector locality in vector interleaving have too small of an effect in dense matrix operations.

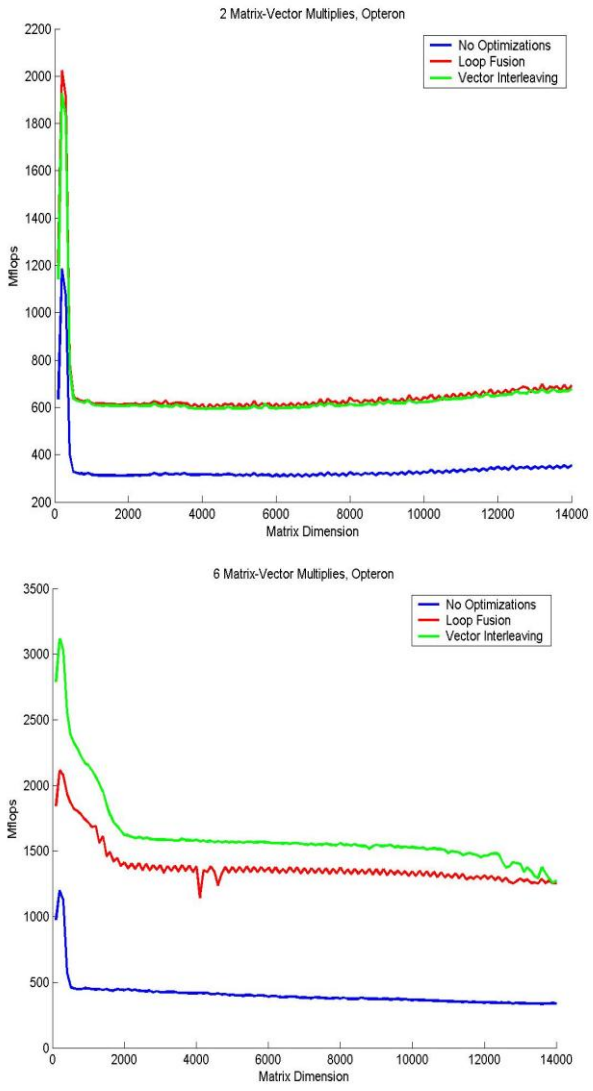

Figure 10: Performance of memory optimizations for increasing matrix sizes, using dense matrix operations

Sparse Matrices

When optimizing sparse matrix operations, vector interleaving provides a performance increase over loop fusion. Since storing sparse matrices uses a similar amount space as the dense vectors in the operations, optimizing for more efficient vector accesses as well as efficient matrix accesses provides significant performance gains. Additionally, loop fusion assumes a linear access pattern into the matrix while, for a sparse matrix, even for a simple operation like a matrix-vector product, the access pattern is somewhat random and unpredictable without knowing the form of the matrix beforehand. Vector interleaving enforces some locality as compared to loop fusion which guarantees some speedup. Figure 11 shows how vector interleaving, the blue squares, is consistently faster than loop fusion, the red diamonds, and than not optimizing, the green circles. The points are all so scattered because the matrices used for the tests are sparse matrices gathered from the University of Florida Sparse Matrix Collection[13], and have many different forms that affect performance.

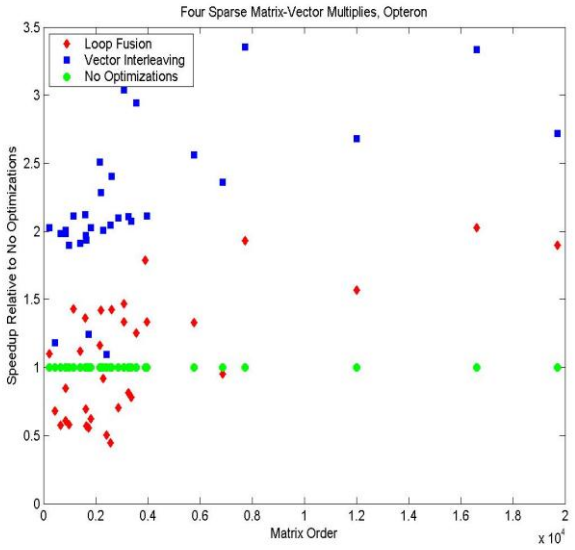

Figure 11: Performance of memory optimizations for increasing matrix sizes, using sparse matrix operations

The reason for vector interleaving performing better is best shown through an example. In Figure 12, we assume that we have two matrix-vector products using a 3x3 matrix, a small cache with only two cache lines, and the matrix is stored in CSR format. If the products are fused, then the first vector is pulled into the first line of cache to read the first element (Figure 12A), and then to read the first element of the second vector the second vector is pulled into the second line of cache (Figure 12B). If the matrix has a form such that the next element needed from both vectors is not something that fit into the cache, in this particular example it is the third element from the vectors, both cache lines would be replaced (Figure 12C and

Figure 12D). If instead the products are interleaved, then the interleaved vector would be pulled into the first line of cache to read the first element, but because of interleaving the second element is the next needed element for the operation (Figure 12E). As such, when the third element from the vectors is needed, they are pulled into the second cache line, and the overall result is only two cache misses as compared to four from loop fusion (Figure 12F). This also shows how vector interleaving enforces some vector locality and cache reuse, since we know that for any element we access in the first vector we will access the same position element in the second vector, and the interleaved vector has those two elements right next to each other. As such, vector interleaving performs better than loop fusion for sparse matrix operations.

Original Vectors

A1	A2	A3
B1	B2	B3

Empty Cache

Line 1		
Line 2		

Matrix

1	0	0
0	0	1
0	1	0

Interleaved Vectors

A1	B1	A2	B2	A3	B3

A: Cache from fusion after accessing A1

Line 1	A1	A2
Line 2		

B: Cache from fusion after accessing B1

Line 1	A1	A2
Line 2	B1	B2

C: Cache from fusion after accessing A3

Line 1	A3	
Line 2	B1	B2

D: Cache from fusion after accessing B3

Line 1	A3	
Line 2	B3	

E: Cache from vector interleaving after accessing A1 and B1

Line 1	A1	B1
Line 2		

F: Cache from vector interleaving after accessing A3 and B3

Line 1	A1	B1
Line 2	A3	B3

Figure 12: In depth example for why vector interleaving performs better than loop fusion for sparse matrices

5. BTO

Since a library of common linear algebra operations is either not complete enough to solve the memory inefficiency issue or is unfeasible to create, we have created a compiler called Build to Order (BTO). BTO accepts MATLAB style syntax as an input, and outputs C code optimized mainly through loop fusion. Additional optimizations are partitioning, cache blocking, multi-threading, and parallelization. Most of the time that BTO spends on optimizing a kernel is spent on finding the ideal set of loop fusions[10]. Since the ideal loop fusion is not always the one with everything fully fused, BTO has to enumerate all possible fusions and find which is best.

BTO works by first checking the input for correct syntax, and creating a graph from all the inputs, outputs, and operations in the given MATLAB style input. In this graph, the vertices are all the inputs, outputs, and operations, while the edges are simply which operations are connected. Once the graph is built, BTO analyzes the different data types used to make sure that all data dependencies are enforced when optimizing and writing code (Figure 13). After these three steps, BTO begins to optimize the code, this is the loop in the rectangle marked "Optimization" in Figure 13. BTO analyzes the graph to create potential new loop fusion combination candidates, generates C code of the new version, and then tests the speed of the code. Depending on the search strategy and the performance of the candidate code, BTO either repeats the process until a better version is found or outputs the candidate as the optimized version. Additionally, if the compiler is run with the -c flag then the correctness test is run at every iteration

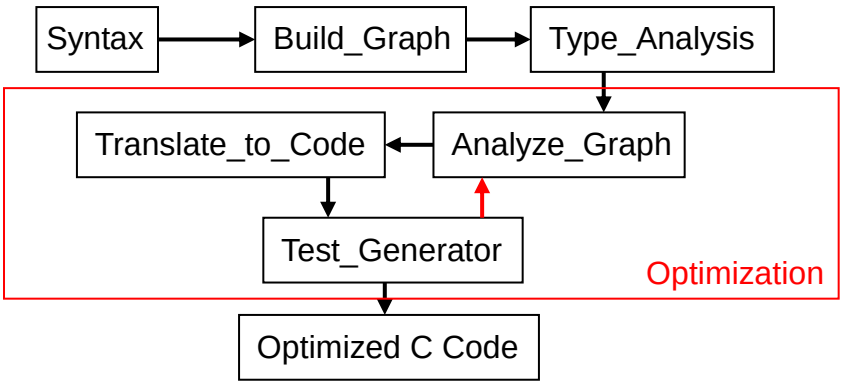

Figure 13: Inner working of BTO

after the Test_generation step. The correctness test is done by running all the given operations using BLAS calls and then comparing the outputs from the BLAS to the BTO versions, ensuring that the norm of the total difference is less than a particular value. This error bound depends on the operations in the given kernel, where every operation has a particular error bound of its own as defined in the IEEE standard. In this section I discuss my additions to BTO.

Division, Norm, and Square Root

I have added the following operations to BTO: square root, division, the norm. All of these are additions in order to make BTO encompass more possible linear algebra kernels. With these additions algorithms like Householder Bidiagonalization can be implemented within BTO. Here, division means dividing every element of a matrix or a vector by a scalar, and not what it means in MATLAB where it is possible to divide a matrix by a matrix, since that is actually an inverse of a matrix. Square root is needed in order to be able to implement the norm, and the norm is commonly used for normalizing vectors along with division. In implementing division and square root, all of the major steps in Figure 13 needed updates. The syntax for square root is "squareroot(x)" where x is the value being square rooted, and the syntax for division is the "/" symbol with the first input on the left and the second input on the right. I introduced new nodes for the graph along with rules for how the nodes should handle inputs and outputs during Type_Analysis. Square root takes in a scalar input and outputs a scalar. Division takes two inputs: the first can be a scalar or a vector and the second can only be a scalar. The last updates are to code generation, test generation, and the correctness tester. This was the biggest update in implementing division, since code generation, test generation, and the correctness tester all have to be able to handle the two different types of division: a scalar divided by a scalar and a vector divided by a scalar, and when a vector is divided by a scalar loop generation is involved. Square root and division are both implemented through the C math library.

The norm has a simpler implementation. The syntax for the norm is "|x|" where x is the vector being normalized, and during the Syntax step in Figure 13 the norm is converted to the square root of a dot product. This means that the norm is applicable only to vectors and not to matrices.

In order to ensure correctness of the implementations, I first used the compiler on several small examples that I can check by reading the code that BTO outputs for correctness. After ensuring that for the small examples the code is properly functional, I ran tests on a bigger set of kernels, ensuring that all the correctness tests from BTO were successful. BTO automates this process by running itself with the correctness checking flag on all given kernels.

Leading Dimension

When a matrix is stored in memory, although the matrix has a row and column index it is stored as a single consecutive block in memory. The C language is row major, so matrices are stored row by row. In comparison Fortran is column major, which means that matrices are stored column by column. In C, to access the i^{th} row and j^{th} column of a matrix we access the i*n_col+j element of the block in memory, where n_col is the number of columns in the matrix. For a submatrix of the matrix, an access to the i^{th} row and j^{th} column is i*ld+j element of the block in memory, where ld is the leading dimension which is equal to the number of columns of the original matrix. The leading dimension, also known as LDA (Leading Dimension of A), allows accesses and operations on submatrices by describing how much bigger the matrix is as compared to the submatrix. Algorithms like Householder bidiagonalization use submatrices to reduce the number of operations.

I implemented LDA in BTO to always be turned on, and the LDA is used when calling the optimized code that BTO outputs. This means that the only way the LDA can be used is by the user calling an optimized kernel on a submatrix. Such a change in the inputs for functions that BTO outputs means that I updated the correctness tester to ensure that the code works for different possible combinations of matrix dimensions and leading dimensions. BLAS implementations handle leading

dimension, so the rest of the correctness tester needed minimal changes. After implementing all of these changes, I first tested by hand whether the LDA was correctly implemented and whether the output functions from BTO could be called by other functions with no errors on small examples. These tests were all successful, so I ran BTO on the full kernel set with the LDA. BTO passes all of the normally functioning tests.

Householder in BTO

Even with the implementation of square root, division, the norm, and leading dimension, certain parts of Householder Bidiagonalization have to be added by hand after BTO optimizes the code. Looking back at the algorithm for calculating a Householder vector (Figure 14), there are several conditionals as well as accesses to particular elements of vectors, neither of which BTO can handle. I took everything in the Householder vector algorithm other than that and had BTO optimize that, and then inserted the necessary conditionals and vector accesses to work the algorithm.

```
Input: x (vector)
Output: v (vector) , β (scalar)
n = length(x)
sig = x(2:n)^T x(2:n)
v = [ 1 x(2:n) ]
if sig = 0
        β = 0
else
        μ = sqrt(x(1)^2 + sig)
        if x(1) <= 0
                v(1) = x(1) - μ
        else
                v(1) = -sig/(x(1) + μ)
        end
        β = 2v(1)^2/(sig + v(1)^2)
        v = v/v(1)
end
```

Figure 14: Algorithm for finding the Householder Vector

Looking back at the rest of the Housholder Bidiagonalization algorithm (Figure 15), we can optimize applying a Householder vector from the left and the right side in BTO. The line right before the if statement and the command before the first end are both storing the Householder vectors in the zeroed out

$$
\begin{aligned}
&Input: A(matrix)\\
&Output: B(matrix)\\
&for \quad j = 1:n\\
&\quad [v, \beta] = house(A(j:m), j))\\
&\quad A(j:m, j:n) = (I_{m-j+1} - \beta v v^T) A(j:m, j:n)\\
&\quad A(j+1:m, j) = v(2:m-j+1)\\
&\quad if \quad j \le n-2\\
&\qquad [v, \beta] = house(A(j, j+1:n)^T)\\
&\qquad A(j:m, j+1:n) = A(j:m, j+1:n)(I_{n-j} - \beta v v^T)\\
&\qquad A(j, j+2:n) = v(2:n-j)^T\\
&\quad end\\
&end
\end{aligned}
$$

Figure 15: Householder Bidiagonalization

portions of the bidiagonal matrix, and these steps BTO can not handle because it has no way to access a vector within a matrix. Because of this, the loops and storing the vectors in the matrix have to be inserted when putting all the pieces optimized by BTO together. This is not the most ideal version of the algorithm to optimize by BTO since the updates to the matrix A are done in two places at separate times, instead of the way that GEMVER would do it in a single place, but it is the simplest form of the algorithm because I want to avoid optimizing the algorithm by hand as much as possible and let BTO handle all of the optimization. Additionally, the other potential slow-down in this implementation is the first step in the for loop, where the algorithm calculates the Householder vector for a column. This is an inefficient step because C is row major and BTO has no implementation of leading stride for a vector, so a temporary vector has to be made and a column of the matrix A is copied into the temporary vector, and then passed into the Householder vector the algorithm. The inefficiency of this step comes from accessing the matrix A across the columns, but this step is only a vector operation so it is not detrimental to the efficiency of the code.

I perform two types of tests to ensure that the implementation is correct. The first is a test by hand, since on a small enough example I can run Householder Bidiagonalization by hand and compare the results. I performed two tests on 5x5 matrices like this, and all numbers matched up perfectly. The other test for correctness is done by doing the updates on the original matrix by all the Householder vectors at once, meaning that the full Householder vectors are used and not the versions that deal only with the submatrices, and comparing the results. This test checks for correctness since if any of the Householder vectors or updates were incorrect, then the final result from this would not be a bidiagonal matrix. I automated the test to run on large matrices and check that the diagonal and super diagonal terms are not all zeroes, and all the off diagonal terms are zeroes. Not all of the diagonal and super diagonal terms have to be non-zero, since if for example the input matrix has a row of all zeroes, then in the bidiagonal form it will also have a row of all zeroes since that portion is already bidiagonalized.

I ran a series of tests on the version optimized by BTO and on DGEBRD, which is Householder Bidiagonalization from LAPACK. LAPACK and DGEBRD were installed using f2c, which is a converter from Fortran to C. All the code for the version optimized by BTO was run with the -O3 flag in gcc. In Figure 16, we can see the results of the performance test, where every point is the average of five runs. BTO performs worse than DGEBRD up until an 800 by 800 matrix is bidiagonalized.

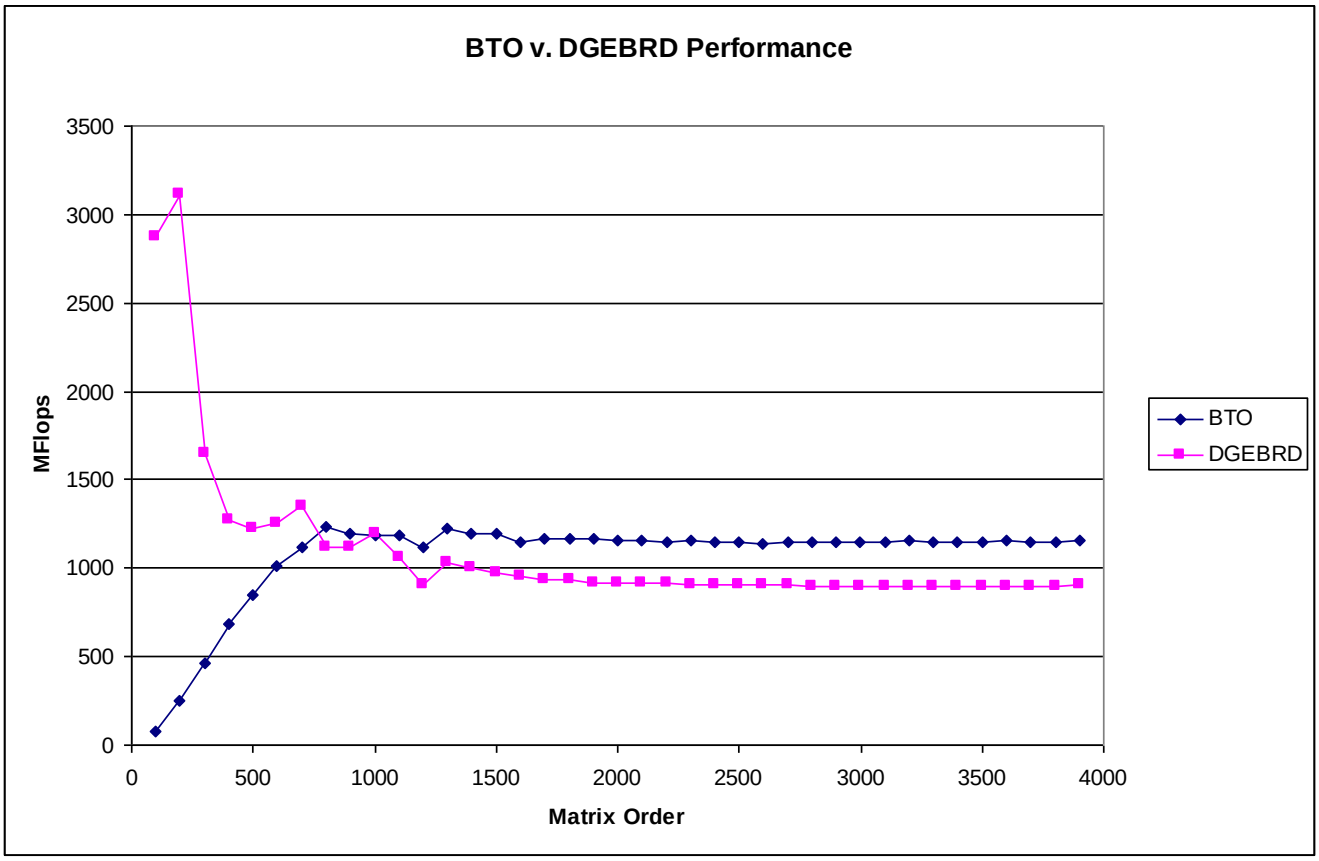

Figure 16: Performance of different Householder Bidiagonalization versions

Beyond that, the version made by BTO consistently performs better than DGEBRD. Householder Bidiagonalization in BTO stabilizes at 1150 MFlops, while DGEBRD stabilizes at 900 MFlops. BTO performing so slowly for small matrix sizes is expected, since loop fusion is an optimization for efficient memory accesses. When a matrix is small, it can easily fit into one of the caches, meaning that loop fusion does nothing. Only once the matrix is large enough to not fit into cache does loop fusion

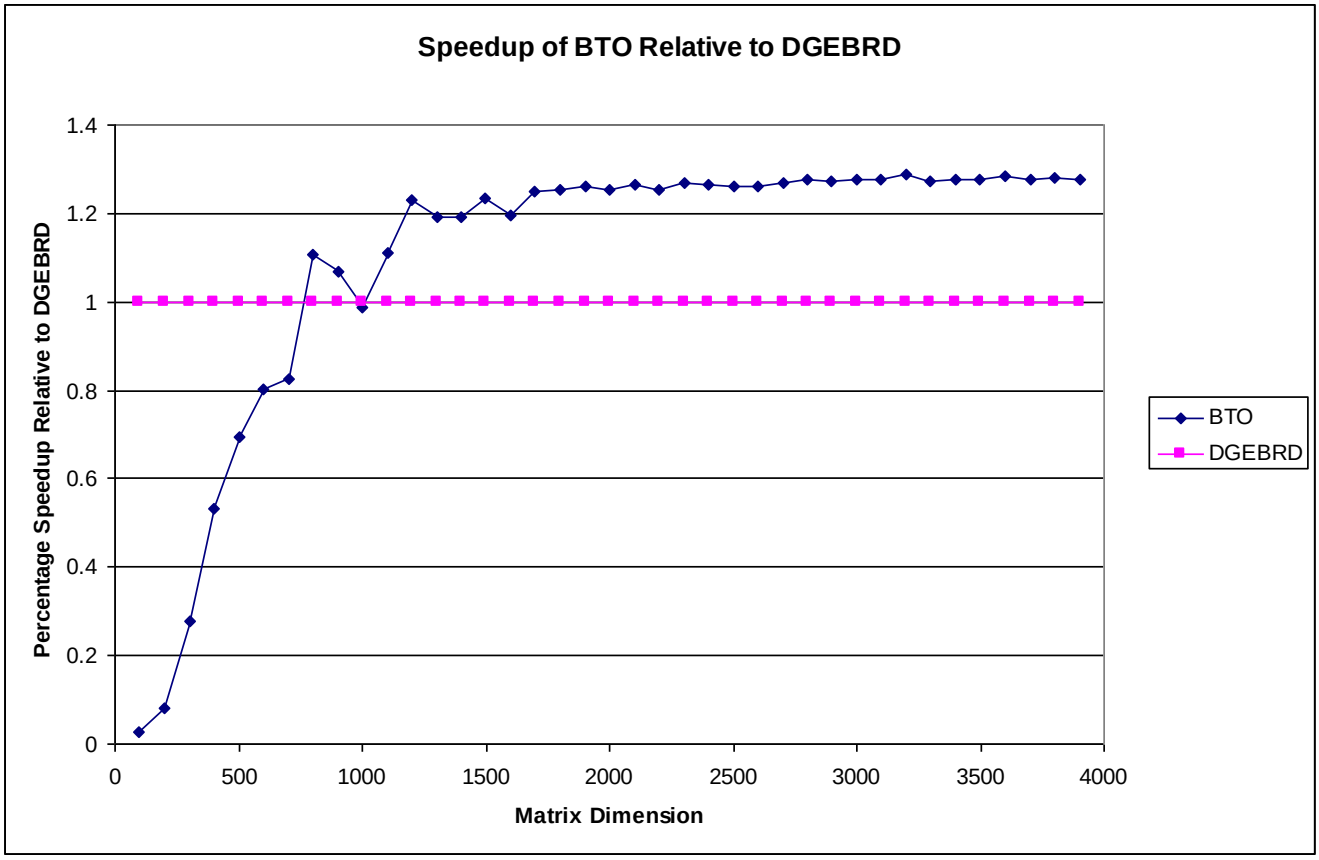

Figure 17: Performance of BTO relative to DGEBRD

show any noticeable performance gains. Figure 17 is the same graph, but the lines are normalized to the performance of DGEBRD. We can see that BTO consistently performs 20 to 30 percent faster than DGEBRD for matrices larger than 1500 by 1500. For very large matrices, the performance increase stabilizes at around 28 percent. In the cache efficient version optimized by using GEMVER and GEMV, Howell reported speedups of up to 32 percent as compared to DGEBRD[5]. Comparing the percentages reported by Howell's results and my results makes little sense though, since we are using

completely different hardware and the libraries I use are not as fully optimized as possible given that I used the f2c version of LAPACK and DGEBRD. Either way, this shows that BTO performs better for Householder Bidiagonalization than a library call, and although it is comparable in performance to hand optimized versions, BTO definitely performs slower than hand optimized versions.

6. Conclusion

I have added a fair amount of extra functionality to the BTO compiler. The biggest and most important addition to the compiler is LDA. Most efficient matrix factorization algorithms involve accessing submatrices in order to limit the number of needed operations. This trick reduces the number of matrix accesses and can at least half the number of operations. As such, the trick is used whenever possible in linear algebra, and is an addition that allows the compiler to do more and gives way for further optimizations using BTO. This is also why it is best to always have the LDA turned on in BTO.

The next most important result is the speedup of Householder Bidiagonalization when using BTO as compared to DGEBRD. This is the first time that BTO has been used to optimize a particular algorithm that solves a problem, before BTO was only tested on kernels composed of a small number of matrix vector operations, and the biggest kernel ever optimized was GEMVER. This result shows that BTO not only works on practical algorithms, but can additionally provide speedups over linear algebra libraries.

Adding division, square root, and the norm is next useful result. The benefits from adding these operations is similar to the LDA, since all the operations expand the number of possible algorithms to implement within BTO. But none of these operations give the same potential speedup that can be gained from using the LDA to the fullest extent. These operations are basic operations that would be expected in such a compiler, but not something that makes the compiler more powerful.

The last useful result is my study of vector interleaving. This result is least useful because it does not directly affect the current compiler. The only time that this result will be useful is when sparse data structures and sparse operations are implemented in BTO.

Overall, I have greatly expanded the functionality of BTO by adding new operations to the compiler. By doing so I was able to expand the number of possible linear algebra kernels that can be implemented within BTO, including Householder Bidiagonalization, which I was able to implement and compare to a library version to prove the efficiency of BTO.

7. References

[1] Gene H. Golub & Charles F. Van Loan. *Matrix Computations*, 3rd Ed. 1996 Johns Hopkins, Maryland.

[2] I. Karlin, E. R. Jessup, G. Belter and J. G. Siek. *Parallel Memory Prediction for Fused Linear Algebra Routines*. In the 1st International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computing Systems (PBMS 10), New Orleans, LA, November 2010, pp 1-8.

[3] John L. Hennessy & David A. Patterson. *Computer Architecture A Quantitative Approach*, 4th Ed. 2007 Elsevier, Oxford.

[4] P. Soderquist, M. Leeser. *Division and square root: choosing the right implementation*. Micro, IEEE , vol.17, no.4, pp.56-66, Jul/Aug 1997.
doi: 10.1109/40.612224

[5] G. W. Howell, J. W. Demmel, C. T. Fulton, Sven Hammarling and Karen Marmol. *Cache Efficient Bidiagonalization Using BLAS 2.5 Operators*. ACM Transactions on Mathematical Software, vol. 34 issue 3, May 2008.
doi: 10.1145/1356052.1356055

[6] David Goldberg. *What Every Computer Scientist Should Know About Floating-Point Arithmetic*. Computing Surverys, March 1991.

[7] A. Silberschatz, P. B. Galvin, G. Gagne. *Operating System Concepts*, 8th Ed. 2009 John Wiley & Sons, Jefferson City.

[8] *The Flagship High-Performance Computing Math Library for Windows, Linux, and Mac OS X*. http://software.intel.com/en-us/articles/intel-mkl/, 2012

[9] K. Goto, R. A. van de Geijn. *Anatomy of High-Performance Matrix Multiplication*. ACM Transactions on Mathematical Software 34(3): Article 12, May 2008.

[10] I. Karlin, E. R. Jessup, G. Belter, J. G. Siek. *Parallel Memory Predictions for Fused Linear Algebra Kernels*. 1st International Workshop of Performance Modeling, Benchmarking and Simulation of High Performance Computing Systems, New Orleans, November 2010.

[11] W. D. Gropp, D. K. Kaushik, D. E. Keyes, B. F. Smith. *Toward Realistic Performance Bounds for Implicit CFD Codes*. Proceedings of Parallel CFD'99, New York, 1999.

[12] W. D. Gropp, D. K. Kaushik, D. E. Keyes, B. F. Smith. *High-Performance Parallel Implicit CFD*. Parallel Comput, 27(2001), pp. 337-362.

[13] T. A. Davis, Y. Hu. *The University of Florida Sparse Matrix Collection*. ACM Transactions on Mathematical Software, November 2010.

[14] Y. Saad. *Iterative Methods for Sparse Linear Systems, 2nd* Ed. Society for Industrial and Applied Mathematics, 2003.

[15] E. Silkensen. *Enhancing the Automatic Generation of Fused Linear Algebra Kernels*. The International Conference on High Performance Computing, Networking, Storage, and Analysis, November 2009.